Maximizing Data Throughput on the EX10xxA Series

INTRODUCTION

The EX10xxA is a family of high-performance signal conditioning and data acquisition products which may include thermocouple and/or voltage measurement channels on a single instrument. It is an LXI instrument which means that all communication is through a common LAN Ethernet interface. The family of products provides a high level of measurement integrity and channel independence which far exceeds alternative methods which generally involve separate elements for the terminal blocks, signal conditioning, and digitization.

There are two basic methods available to gather data from the EX10xxA besides using the embedded web page to collect data. One method is simply to poll the memory of the instrument at some regular rate. While this method is simple to implement, there are several major problems that users may experience depending on test conditions. One is that it is not possible to collect data fast enough by polling to prevent a memory overflow at higher sampling rates. Such an overflow will result in a loss of data as soon as the 14 MB memory on the EX10xxA is filled. This is unacceptable in many test applications, especially over longer testing periods. A second problem is that constant polling is inefficient for the processor as well as the LAN interface—where the normal implementation is to poll as quickly as possible. For example, even when there is no data present, the instrument still needs to receive the request for data and send a response to indicate that no data is present. It is important to note that LXI is not a strictly deterministic interface and polling at some regular rate does not guarantee that you will receive the same amount of data every time. Figure 1 shows the flow of a typical FIFO polling setup.

DID YOU KNOW?

The EX1000A family implements IEEE-1588 precision time stamp protocol and the LXI wired trigger bus to precisely synchronize data and allow for distributed high-channel count systems.

Find out more about the EX10XXA Series

EX1048A: 48-Channel Precision Thermocouple Measurement Instrument

Find out more about how the EX1048A was used in combination with other VTI products to provide a mission-critical LXI DAQ system

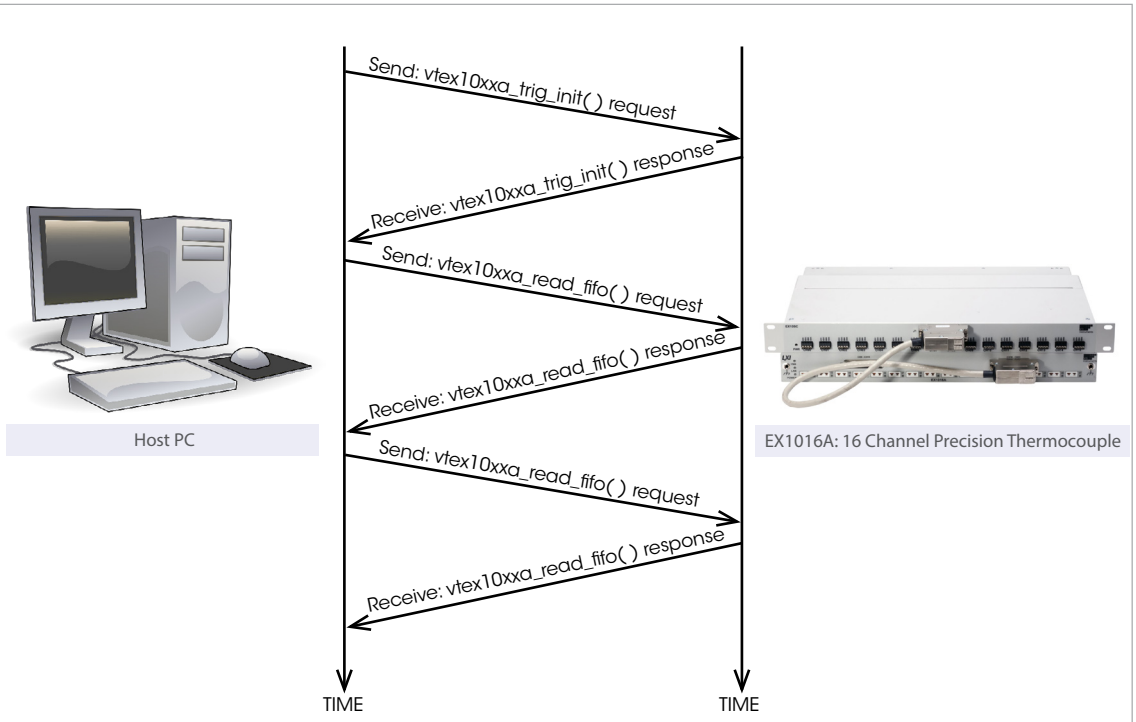

FIGURE 1: READ FIFO POLLING METHOD

The second method, Asynchronous Data Streaming, allows data to be sent as soon as it becomes available and the LXI interface allows, without having to send queries to the instrument after setup. This overcomes the problems with the polling method. It requires two threads to be running simultaneously: A Streaming Server which collects pages of data sent to it asynchronously from the instrument; and an API which sets up the instrument, then polls for data from the Streaming Server thread and processes that data as needed. Contrast the previous figure with the streaming method shown in Figure 2. More data is available at the host PC per unit time.

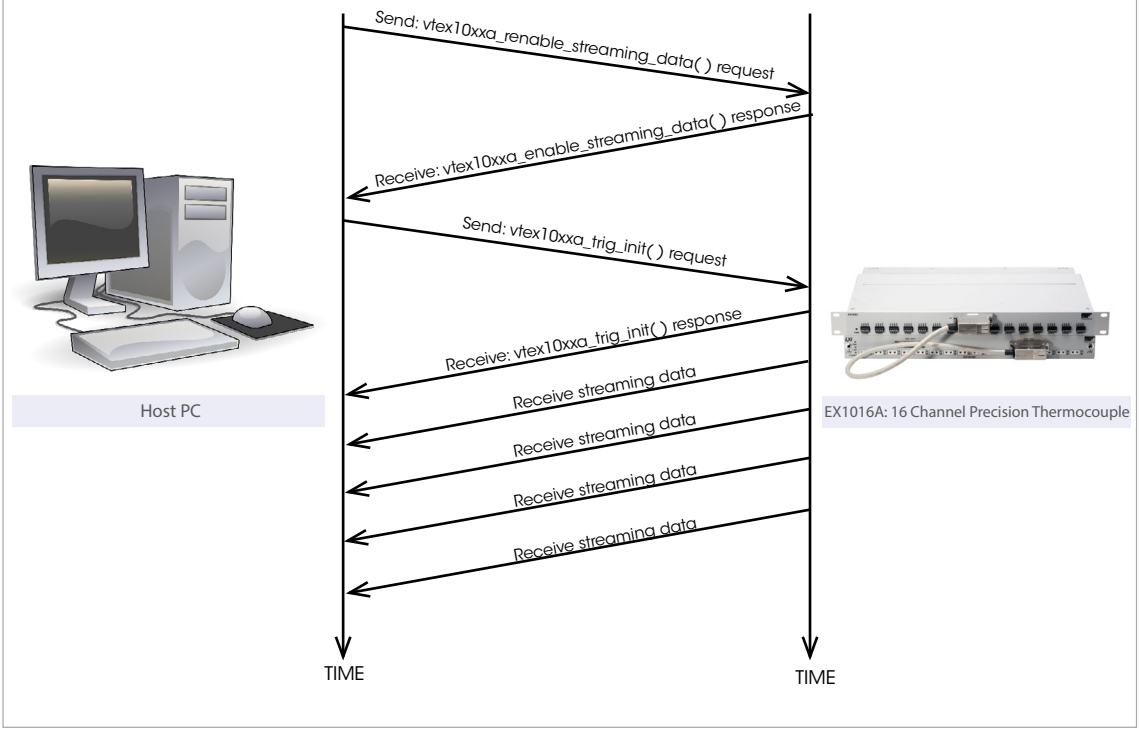

FIGURE 2: ASYNCHRONOUS STREAMING METHOD

SOFTWARE REQUIRED

- Microsoft Visual Studio C# 2010
- VTI Instrument EX10xxA Plug&Play Driver
- Labview 2011 (or later)

STREAMING SERVER

Streaming requires a multi-thread approach in which two processes are essentially running at once. Modern operating systems can take advantage of multiple core processors by simultaneously dedicating process flow to different cores. One such process in this application runs in the background and handles all data received from the instrument by copying it over to a FIFO on the host PC so that it can later be collected and processed by the main API. The actual example source code for both the streaming server and the Labview API should accompany this document and you should refer back to it while reading.

The following exposed functions are basic necessities for creating the simple streaming server application provided in this example.

StartServer—Initializes a Critical Section Object for the callback and allocates memory for the data pages sent by the EX10xxA. This initialization creates a dedicated process which handles the incoming streamed data. For this example, there are no parameters used. Successful memory allocation returns 0.

GetBufferCount—A simple query which returns the number of data pages currently in the Server's FIFO. This basic query does not check the content of any of the data collected.

ReadStreamingData—This is the main accessor function for the Streaming Server's FIFO data. In this example, we are only extracting the timestamp components (ss_secs, ss_nsecs) and sample values (x) members of the EX10xxA data structure. The other members of the data structure are shown below and further detailed in the EX10xxA User's Manual where it documents the EX10xxA_enable_streaming_data function of the Plug&Play driver. We also return an integer called "devId" which can be uniquely assigned to each EX10xxA box which is streaming data in order to identify which instrument the data is coming from.

```
struct DataPage
{
    EX10xxA_SampleData *data;
    int devId;
};

struct EX10xxA_SampleData
{
       ViUInt32 ss_secs; /* data sample time stamp */
       ViUInt32 ss_nsecs; /* fractional part of time stamp in nanoseconds
                             (microseconds in legacy mode) */
       struct {
              ViUInt32 x_len; /* array size of data samples */
              ViReal32 *x_val; /* an array of data samples */
       } x;
       struct {
              ViUInt32 x_counts_len; /* array size of ADC counts */
              ViInt32 *x_counts_val; /* an array of 20-bit ADC Readings */
       } x_counts;
       struct {
              ViUInt32 x_ticks_len; /* array size of x ticks */
              ViUInt16 *x_ticks_val; /* 16-bit measurement timestamps-100 ns
                                        offsets from scan start time */
       } x_ticks;
       struct {
              ViUInt32 cjc_len; /* array size of cjc data*/
              ViReal32 *cjc_val; /* an array of 32-bit floating point CJC
                                    values */
       } cjc;
```

```
        struct {
                ViUInt32 cjc_counts_len; /* array size of cjc count */
                ViUInt32 *cjc_counts_val; /* an array of 20-bit ADC Readings */
        } cjc_counts;
        struct {
                ViUInt32 cjc_ticks_len; /* array size of cjc ticks */
                ViUInt16 *cjc_ticks_val; /* 16-bit CJC timestamps - 100 ns off
                                            sets from scan start time */
        } cjc_ticks;
        struct {
                ViUInt32 limits_len; /* array size of limit data */
                ViUInt32 *limits_val; /* limit data 4 bits for each channel */
        } cjc_counts;
        struct {
                ViUInt32 cjc_ticks_len; /* array size of cjc ticks */
                ViUInt16 *cjc_ticks_val; /* 16-bit CJC timestamps - 100 ns off
                                            sets from scan start time */
        } cjc_ticks;
        struct {
                ViUInt32 limits_len; /* array size of limit data */
                ViUInt32 *limits_val; /* limit data 4 bits for each channel */
        } limits;
};
```

Note that there is only a single two-part timestamp and one devId per scan (page of data) associated with up to 48 channels data per scan. The time stamp indicates when the scan started. If a timestamp for each measurement is desired, the "x_ticks" member can also be exposed.

DeleteDatapage—This function can be used to remove a page of data from the Server FIFO without reading it. This can be useful if a previous scan is interrupted and unread data is left in the Server FIFO. You may want to clear out this data to begin with an empty FIFO. In that case, this function should be looped for each page of data found remaining in the FIFO before a new scan is initiated.

StopServer—This function is necessary to delete and close out the Critical Section process.

Two unexposed functions are also used. ServerCallback is a function which handles the synchronous callback executed whenever data is received by the host PC from the box. It is best to optimize this function as much as possible so that it is ready to handle the next block of data it receives as soon as possible. To accomplish this, it must allocate memory for storing the streamed data and put it on the Server FIFO. CopyDatapage handles the copying of the various data structures that the callback requires.

LABVIEW API

The state machine Labview model was used in this example to break up the various parts of the application.

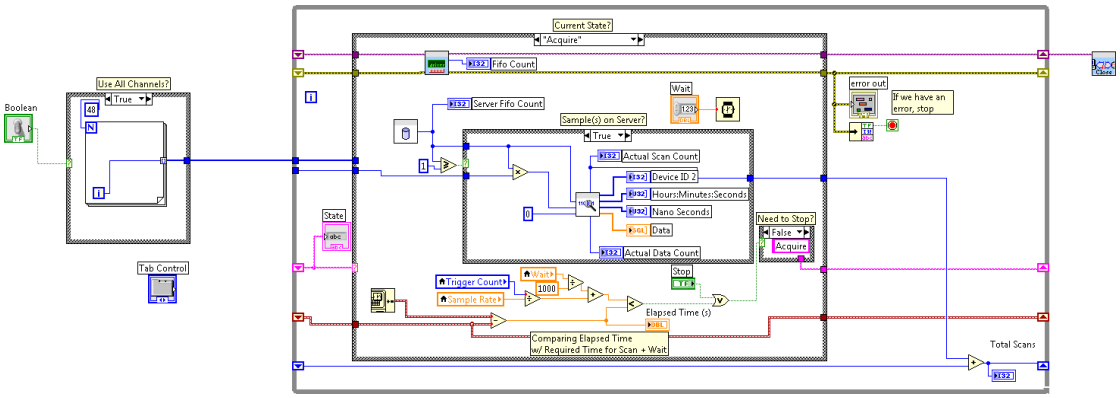

FIGURE 3: OVERVIEW OF LABVIEW API FLOW DIAGRAM

Here is an overview of the states that are used:

1. Init—This default state initializes the card and resets all measurement parameters to the default state.

2. Setup Channels—This state generates the scan list as well as the EU Conversion and Filter frequency for those same channels. Alternatively, a user could have a subset list of channels with a variety of different settings depending on what they are trying to measure. This state also sets the trigger count, which is the number of triggers received after initiating measurement before the unit returns to the idle state. Alternatively, a "Set Trigger Infinite Enabled" VI could be used to enable continuous acquisition. This would override any trigger count setting.

3. Setup Triggering—This state sets up the various aspects of triggering. This simple example uses the default immediate arm setting and the trigger is some decimation of the timer (internal EX10xxA oscillator). This is a simple way to set a uniform scan rate on a box. If multiple boxes are used, some VIs for synchronous arming are recommended here.

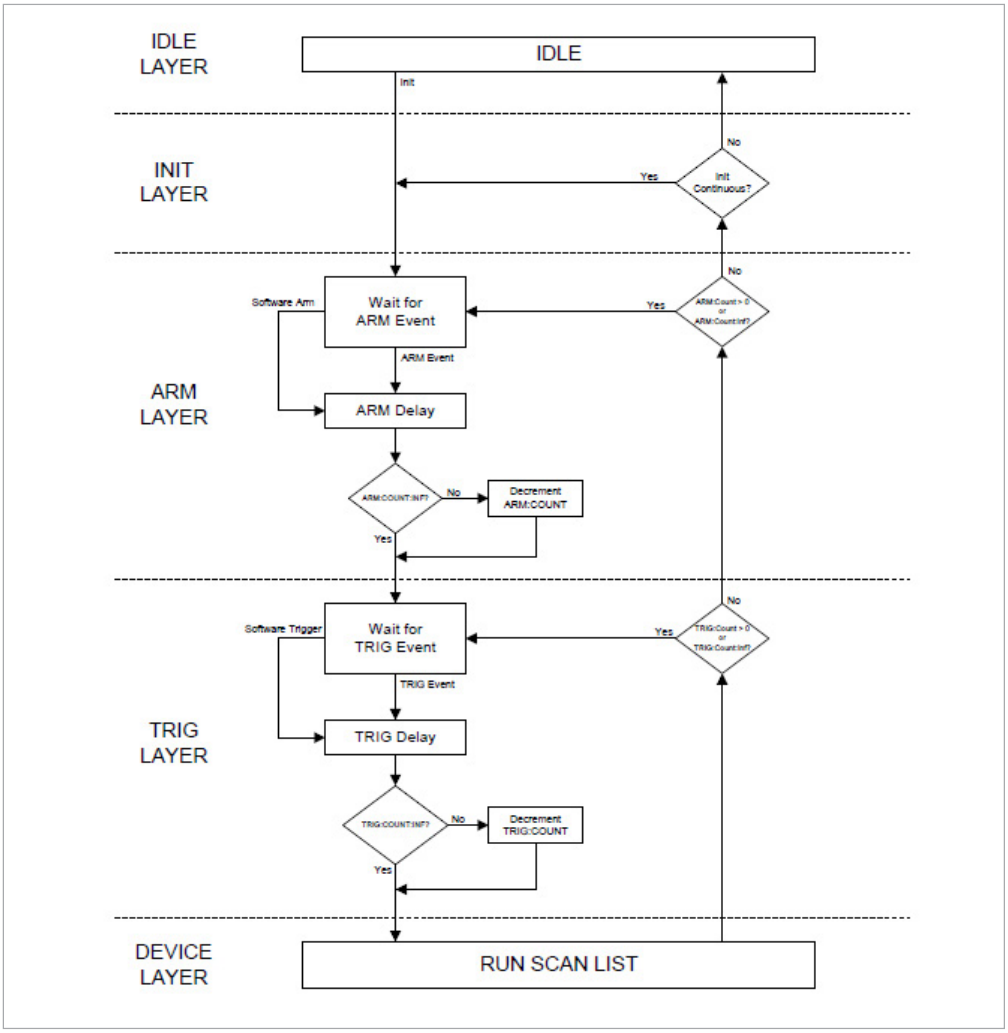

FIGURE 4: EX10XXA TRIGGER STATES

4. Enable Streaming—This state uses a VI built around the streaming server to begin waiting for data streamed from the acquisition system. It then enables streaming on the box. The device ID is an input to the Enable Streaming VI and can be any value. However, if multiple boxes are running you should use a unique value for each one so that you can identify where a given page of data is coming from.

5. Initiate Acquisition—This state advances the trigger model from Idle layer to the Arm layer. Since the arming configuration is set to immediate by default, it then proceeds directly to the Trig Layer. In this example, the decimated timer signal causes a Trig Event which causes a single scan to run on every tick set by the sample rate. After Acquisition is initiated, the system time is recorded in order to keep track of how long the acquisition has been running.

6. Acquire—This is the state which polls for and displays any data on the Streaming Server FIFO. This state loops continuously until one final reading is made after the acquisition time should have elapsed, or until the user decides to stop the scan via the front panel button.

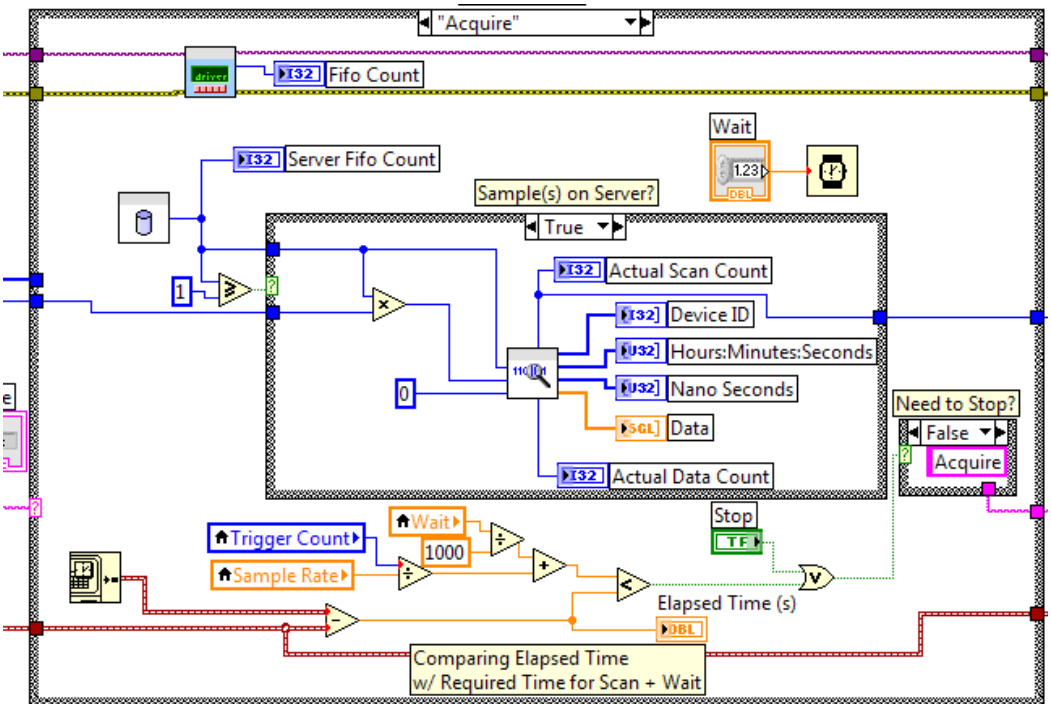

FIGURE 5: LABVIEW API, ACQUIRE STATE DETAIL-

The "Get FIFO Count" VI queries the number of samples remaining on the box, waiting for the LXI interface to send it to the PC. Another VI queries the FIFO on the Streaming Server. The wait control should be set to some time sufficient enough to execute the loop completely. 100ms is generally enough time for even the maximum scan rate.

7. Stop—This state simply shuts off the streaming server after the acquisition has been stopped. Figure 6 displays the front panel controls.

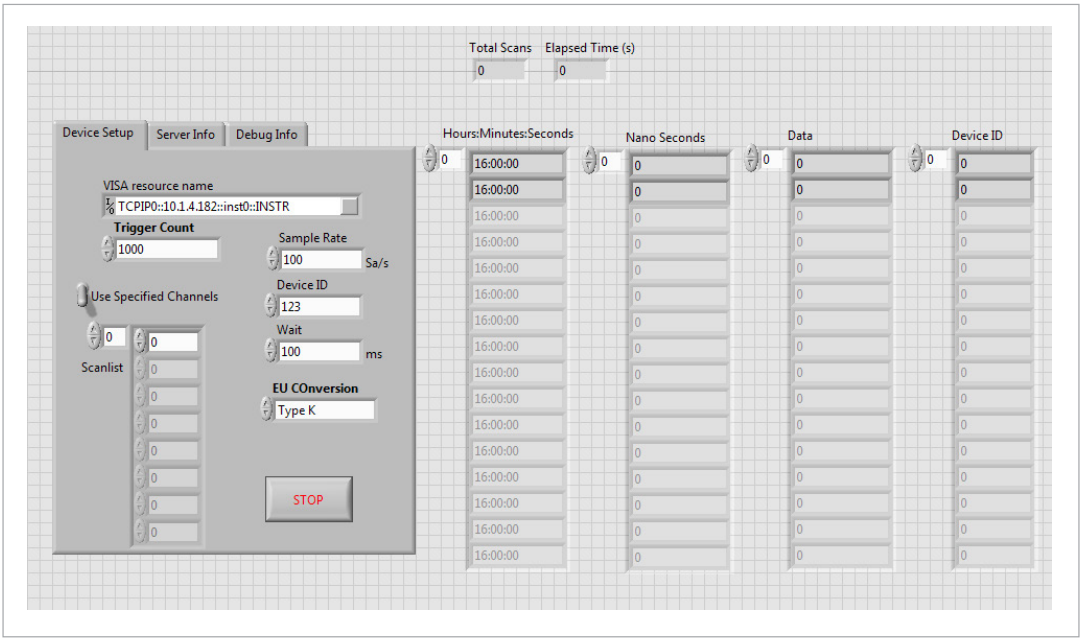

FIGURE 6: LABVIEW API CONTROL PANEL

CONCLUSION

By using Asynchronous Streaming, we overcome the limitations and inefficiencies associated with the instrument polling method of data collection. Data is streamed to the host machine as soon as it is available and collected into a FIFO on the host PC. The main User API then polls this data from the FIFO on the host PC in a much more efficient manner. This allows the User to access the highest sampling rates available on the device without a risk of overflowing the EX10xxA memory. It also represents about half as much network traffic as the polling method would require.